15295 Fall 2017 #9 -- Problem Discussion
October 25, 2017

This is where we collectively describe algorithms for these problems. To see the problem statements follow this link. To see the scoreboard, go to this page and select this contest.

A. Daydreaming Stockbroker
Think in this way: you are allowed to regret and sell the stock with the highest price after you bought it. Then you can always make the optimal moves. O(N) --hpfdf

B. Bacteria Experiment

Consider any vertex *v* in the graph. After one time step, *v* has an edge to any vertex that was originally distance <= 2 away from *v*. Then after another time step, *v* has an edge to any vertex that was originally distance <= 4 away from it. In general, after *t* units of time, *v* has an edge to any vertex of original distance <= 2^*t* away from *v*.

We conclude that the number of time units it takes for all edges to be added to the graph is determined by the largest distance between two vertices in the graph, or the *diameter* of the graph. If the diameter of the graph is *D*, then the answer is ceiling(log_2 *D*).

We can find the diameter of a tree in *O(n)* time.

One way of finding the diameter in *O(n)* time: Take any vertex *r*. BFS from *r* to find the farthest vertex *v* from *r*. BFS again from *v* to find the farthest vertex *u* from *v*. It turns out that the distance between *v* and *u* is maximal. This is sort of annoying to prove, but you can sit down and work it out, or look up the proof online. (For the proof... think of the tree as consisting of the path from *u* to *v*, plus a bunch of stuff that branches off of this main path. If there are a pair of vertices that have larger distance apart than *u* and *v*, what does the tree look like then, and what happens to the BFSs? Drawing pictures may be helpful.)

Another way of finding the diameter, which might be a bit easier to conceptualize: Root the tree arbitrarily. It's not hard to see that the longest distance in the graph will be achieved by a pair of leaves. For each vertex *v*, compute its height *h*(*v*) (the length of the longest path from *v* to a leaf). Then for each *v*, we can compute the longest path between pairs of leaves whose LCA is *v* (it's h(u) + h(w) + 1 where u, w are the two heightiest children of *v*). Take the max of this over all *v* to get the longest path overall. This can all be done with one or two DFSs.

-- Tom

C. Emptying the Baltic
BFS from the draining point to find out the drainable altitude of each grid. I used a priority queue (by minimal draining altitude). This ensures O(N^2logN) time. --hpfdf

D. Fleecing the Raffle

Suppose you put your name x times into the box. There are (n+x) names, and p will be chosen. The probability of your name being chosen exactly once is

x * C(n, p-1) / C(n+x, p)

which can be simplified to

xp/(n-p+1) * Product[i=1 to x; 1 - p/(n+i)]

We can then enumerate x to find the max answer. The target starts to monotonic decrease after x is larger than best answer. An early break makes this algorithm O(N). --hpfdf

Doing some math, we see that the above expression (x * C(n, p-1) / C(n+x, p)) is maximum at x = n/(p-1). So, just computing the above product for this particular value of x is also sufficient. -- Shreyan

E. Compass Card Sales

Construct a BST for cards in each color channel by their channel values. The BST must be able to find prev/next node. To resolve the cyclic adjacent issue, make the largest node = the prev of the smallest node, and vice versa.
Build another priority queue (or BST) by their calculated uniqueness. Each time after printing and removing the most unique card, use the channelwise BST's to find out whose uniqueness should be updated after the removal. There should be at most 6 affected cards (two adjacents in each of three channels). So this algorithm is O(N log N). --hpfdf

F. Highest Tower

Build an undirected graph as follows. Each rectangular dimension defines a vertex. So if a rectangle is a×b with a ≠ b then there are two vertices, one labeled a, the other b. Connect these two vertices with an undirected edge. (Use a hash table to renumber them from 0 to nn-1.) So there is an edge from i to j if rectangle (i,j) exists. Note that if it exists twice there are two edges between them. (A multi-edge.)

A valid tower corresponds to assigning an orientation to each edge in this graph such that the in-degree of each node is at most one. And this defines which way we will use that rectangle (edge). The dimension to which the arrow points is the horizontal direction for that rectangle (edge). Because the in-degree of a vertex is at most one, it's guaranteed never to use two rectangles with the same dimension horizontally. Thus the tower is possible.

Now each component of this graph is either a tree or a graph with one cycle in it. If it has more than one cycle then the tower is impossible to build -- in any orientation of the edges in this case must result in some vertex with in-degree two or more.
Our job is to orient the edges satisfying the above in-degree ≤ 1 condition and also maximize

Sum (over all vertices v) of out-degree(v) * value(v)

We process each component separately. If the component is a tree we just do a DFS starting from the vertex of highest value (a high-value node wants a high outdegree). And in the DFS tree we orient edges from parent to child.

If the component has a cycle we again do a DFS, but this time starting from any vertex on the cycle. The edges are oriented again in the same fashion (parent to child in the DFS spanning tree). We need to make sure to include the edge that closes the cycle.

The algorithm runs in O(n) time. ---DS

Implementation note: There is a very nice way to implement an undirected graph which supports multi-edges and DFS. The edges are numbered. If e is an edge, maintain an array vertex[e] which stores the sum of the vertex numbers at the two ends of this edge. Every vertex also stores a list of all the edges incident on it. Now when we are doing a DFS, we pass in the vertex we've reached along with the edge we used to get there. So it's DFS(v,e). When scanning the adjacency list of v we skip edge e (we don't want to follow it backwards back to our parent). For another edge f from vertex v, we call DFS(vertex[f]-v, f). This gives a very elegant way to handle multi-edges, which are needed in this problem, and also to prevent the DFS erroneously going backwards. I didn't figure this out until I had already implemented a much more complex solution.

G. Exponial

Lemma: for all n and m and b >= $log_2 m$, then

$$n^b \% m = (n\%m)^{\phi(m) + b\%\phi(m)} \% m$$

where $\phi(m)$ is Euler's totient function and %is the modulo operation.

Let f(n,m) = exponial(n) % m then applying this lemma we know that

$$f(n, m) = (n\%m)^{\phi(m) + f(n-1, \phi(m))} \% m$$

Notice that any number mod 1 is 0. So we can basically recurse until n goes below 5 (where we calculate exponial(n) % m directly) or $\phi(m)$ becomes 1.

For computing the powers, we can use quick power algorithm that runs in $O(logm)$ time.

So now the problems boils down to find $\phi(m)$.

$$\phi(m) = m\prod_{p|m}(1 - \frac{1}{p})$$

We can just use a $O(\sqrt{m})$ factorization algorithm to find all the prime factors of m and then compute $\phi(m)$ by the above equation.

Let T(n,m) denotes the runtime of f(n,m) then we know

$$T(n, m) = T(n - 1, \phi(m)) + O(\sqrt{m})$$

So we know $T(n, m) = O(\phi^*(m)\sqrt{m})$ where $\phi^*(m)$ is the number of times we need to apply the totient function to get n to 1.

Now we claim that $\phi^{*}(m) \leq logm$. And we can prove this by induction noticing the fact that $\forall_{m}$, m is even, $\phi(m) \leq \frac{m}{2}$.

Thus, this algorithm runs in $O(logm\sqrt{m})$ overall. ---Tim